Why PowerShell?

by Warren Frame
and Don Jones

PowerShell.org

Table of Contents

ReadMe	0
About this Book	1
A Brief Overview	2
Why Scripting? Why a Shell?	3
Why PowerShell?	4
The Business Story	5
Where Can You Learn More?	6
Why PowerShell Remoting? (While we're answering "Whys")	7

Ever wonder why PowerShell is a thing, or find yourself having to explain it to someone else? Here's a concise guide that does just that.

Why PowerShell?

By Warren Frame and Don Jones

An incredibly concise look at why Windows PowerShell is important, from both a technical and business perspective.

This guide is released under the Creative Commons Attribution-NoDerivs 3.0 Unported License. The authors encourage you to redistribute this file as widely as possible, but ask that you do not modify the document.

Was this book helpful? The author(s) kindly ask(s) that you make a tax-deductible (in the US; check your laws if you live elsewhere) donation of any amount to The DevOps Collective to support their ongoing work.

Check for Updates! Our ebooks are often updated with new and corrected content. We make them available in three ways:

- Our main, authoritative GitHub organization, with a repo for each book. Visit https://github.com/devops-collective-inc/
- Our GitBook page, where you can browse books online, or download as PDF, EPUB, or MOBI. Using the online reader, you can link to specific chapters. Visit https://www.gitbook.com/@devopscollective
- On LeanPub, where you can download as PDF, EPUB, or MOBI (login required), and "purchase" the books to make a donation to DevOps Collective. You can also choose to be notified of updates. Visit https://leanpub.com/u/devopscollective

GitBook and LeanPub have slightly different PDF formatting output, so you can choose the one you prefer. LeanPub can also notify you when we push updates. Our main GitHub repo is authoritative; repositories on other sites are usually just mirrors used for the publishing process. GitBook will usually contain our latest version, including not-yet-finished bits; LeanPub always contains the most recent "public release" of any book.

A Brief Overview

PowerShell etnhusiasts often find themselves explaining why someone with responsibilities on the Microsoft side of the IT shop should learn PowerShell. We decided to write this as a reference going forward.

We won’t be arguing for PowerShell over other Microsoft languages such as VBScript or batch, or general purpose languages such as Python or Perl. There is a place for all of these languages, but if you work with the Microsoft and surrounding ecosystems, PowerShell is an important language to learn.

What's also important to understand is that *Microsoft has made an enormous commitment to PowerShell*. It isn't going away, and indeed the company is building more and more of their management solutions on top of it. To a degree, Microsoft is even backing off from building management tooling, knowing that you can use PowerShell to build your own tools. That's significant.

But let's move on.

Why Scripting? Why a Shell?

Before we dive into PowerShell itself, let's tackle the importance of scripting and automation, an integral facet of PowerShell.

You've probably seen this XKCD comic or something similar to justify scripting. While saving time is certainly a factor behind the importance of scripting and automation, it is hardly the only justification.

Here are a few others to consider:

- **Consistency**. A scripted solution will run the exact same script every time. No risk of typos, forgetting to complete the task, or doing the task incorrectly. *Reduce human error*.
- **Audit trail**. There are many tasks where having an audit trail would be helpful, perhaps including what task was performed, important results, errors that occurred, when the task ran, who ran it, and so forth. Scripts can provide this trail, and in PowerShell v5 and later, the shell itself features extensive logging capabilities.
- **Modular code**. You might spend more time on a particular function than time savings justify, but you can generally re-use or borrow ideas from the code later.
- **Documentation**. Is there documentation for the task? Is it up to date? A well written and commented script can generally serve as a helpful base level of documentation that might not exist for a manual task. In some cases, the script can document the process that it automates, helping to preserve institutional knowledge.
- **Education**. Administrators who can automate tasks are almost always more well-versed in the technology as a result. That makes them better planners, architects, troubleshooters, and operators, all of which convey benefit to the organization.
- **Delegation**. With a scripted solution, you can typically delegate more functions closer to the teams best equipped to handle them. With PowerShell v3 and later specifically, scripts can enable extremely granular delegation of tasks, helping the overall IT team become more efficient and responsive.

The moral of the story is that scripting and automation is important, which is just one factor behind the value of learning PowerShell.

Why PowerShell?

Microsoft describes PowerShell as “a task-based command-line shell and scripting language… built on the .NET Framework.” What is so great about PowerShell? Why should you use it?

PowerShell is both a command-line shell and scripting language

Fight fires quickly using existing or custom PowerShell commands or scripts at the shell, no need to compile code. Develop your code at the command line before creating a function or script around it. Write quick and dirty scripts that you will use a single time or a handful of times. Write formal, readable, production level scripts that will maintain your services for years.

What is the cost of this investment? Learning PowerShell. Pretty reasonable, considering you will likely need to do so regardless of your current language of choice, assuming you work with the Microsoft ecosystem.

PowerShell can interact with a dizzying number of technologies.

The .NET Framework, the Registry, COM, WMI, ADSI. Exchange, Sharepoint, Systems Center, Hyper-V, SQL. VMware vCenter, Cisco UCS, Citrix XenApp and XenDesktop. REST APIs, XML, CSV, JSON, websites, Excel and other Office applications. C# and other languages, DLLs and other binaries, including Linux or Unix tools. A language that can work with and integrate these various technologies can be incredibly valuable.

Windows is not text based. Sooner or later you will need to do something that you can’t do with -nix tools and other text based languages. Many of the technologies that PowerShell can interact with simply do not have text based interfaces, and may not even be directly accessible from more formal languages like Perl or Python.

The moral here is that PowerShell is the best "glue" Microsoft has ever provided us for tying together disparate systems. It's better than previous Windows-based shells because it understands, and works with, the API-based nature of Windows itself, which is vastly different from what previous text-based shells did.

PowerShell is object-based.

This gives us incredible flexibility. Filter, sort, measure, group, compare or take other actions on objects as they pass through the pipeline. Work with properties and methods rather than raw text.

If you have spent time deciphering and programmatically working with text based output, you know how frustrating it can be. What delimiter do I split on? Is there even a delimiter? What if a particular result has a blank entry for a column? Do I need to count characters in each column? Will this count vary depending on the output? With objects, this is all done for you, and makes it quite simple to tie together commands and data across various technologies.

PowerShell isn’t going away.

Microsoft is putting its full weight behind PowerShell.

PowerShell support is a requirement in the Microsoft Common Engineering Criteria, and a Server product cannot be shipped without a PowerShell interface. That means very few Microsoft server products can't be managed by PowerShell - and the few that can't, will be able to soon.

Vendors other than Microsoft have strong support for PowerShell. This includes IBM, Cisco, Citrix, VMware, NetApp, Dell, and dozens more.

In many cases Microsoft uses PowerShell to build the GUI management consoles for its products. Some tasks can’t be performed in the GUI and can only be completed in PowerShell. This is a big deal: In an increasing number of situations, *you can't manage the product fully unless you use PowerShell.* That applies to cloud-based offerings like Azure and Office 365, too.

Consolidate and multiply your learning

Your reward for learning PowerShell is the improved ability to control and automate the many technologies it integrates with. You can use the same set of commands to filter, export, redirect, modify, extend, and perform actions against output for all of these technologies. You can pick up PowerShell skills and take them in any direction you need - Hyper-V, vCenter, SQL, AD, XenApp, and more.

Your reward for learning specific tools or executables such as net.exe or schtasks.exe, is the ability to work with those specific tools. With PowerShell, your learning investment blossoms into an enormous ecosystem of capability.

In Windows, PowerShell's really the only option

VBScript is deprecated, and you're not going to see further development. VBScript was already anemic in terms of the things it could "touch," making it a poor "glue" for connecting systems and processes.

And nothing else even came close to VBScript. Python, Perl, you name it - they're great on Linux, a predominantly text-based OS, but they were nigh-useless on Windows, since they couldn't access the many and varied APIs Windows uses for management.

PowerShell consolidates all of those APIs into a single, largely-consistent interface that's focused on systems operations and administration.

The Business Story

If you've ignored PowerShell up until now, or were skeptical about it, let's look at what Microsoft has done.

In **version 1**, PowerShell emerged as a the first management interface specifically designed for administrative automation.

In **version 2**, PowerShell gained native remote management capabilities, enabling remote management of any server or client running PowerShell. PowerShell's "reach" extended to hundreds of management APIs, enabling real-world management. The product also matured a deceptively simple, powerful scripting language that can be used to build professional-grade units of automation.

In **version 3**, PowerShell learned to run long-running tasks in a disconnected, stateless fashion - called *workflows*. The product's reach extended even further, covering all major Microsoft server platforms, and pushing into Microsoft's cloud offerings. By this version, PowerShell was a very real thing, so much so that many Microsoft native GUIs began to use PowerShell "under the hood."

In **version 4**, PowerShell was extended with even more "reach," and gained a new technology: Desired State Configuration. DSC lets administrators describe, in more-or-less plain text, how a computer should be configured. Leveraging the existing investment in PowerShell, DSC then puts the machine into that state, and *keeps* it there.

In **version 5**, PowerShell matured DSC and extended its "tool making" capabilities into professional developer space. With support in Visual Studio, PowerShell started to span a much broader spectrum of user, from entry-level administrators to advanced developers.

The point is that Microsoft has *clearly* been building PowerShell since its v1 release in 2006. They've done so in a way that *they've never done before* in languages like VBScript, and they've done so while maintaining consistency and efficiency.

What's more, PowerShell has inspired a broad ecosystem of supporting vendors, and an enthusiastic global community. Administrators are, more than ever, able to get assistance, answers, and even ready-made solutions from those vendors and that community.

Where can you learn more?

There is a wealth of information on PowerShell.

- Start at PowerShell.org, a community-owned and -operated web site that hosts a friendly Q&A forum. The organization also offers numerous free ebooks, runs the annual PowerShell Summit event in North America and Europe, hosts a DSC GitHub repository, runs an annual Scripting Games contest, and much more.
- Anyone with development or scripting experience should pick up PowerShell in Action v2. It's written by the co-designer and principle author of PowerShell, Bruce Payette, and is the standard reference. It provides the best depth you will get short of verbose articles on the web, gives insight into some of the design decisions behind the language, and is perfectly applicable today despite being written for PowerShell v2. Windows PowerShell for Developers is more narrowly focused but a good read for the experienced.
- Those without scripting or development experience might want to start with lighter reading, such as Learn Windows PowerShell in a Month of Lunches.
- Want to learn on the fly? PowerShell includes everything you need to learn directly from the shell. Get-Command, Get-Help, Get-Member, and Select-Object will get you exploring and learning PowerShell.
- Prefer videos? Product inventor Jeffrey Snover and Jason Helmick hosted two free PowerShell sessions that have proven quite popular: Getting Started with PowerShell 3.0 and Advanced Tools and Scripting with PowerShell 3.0. Or, check out the PowerShell.org YouTube channel, featuring technical videos and session records from every PowerShell Summit event.

Why PowerShell Remoting? (While We're Answering "Whys")

Another big question that comes along is, "why should we enable PowerShell Remoting?"

First, understand a couple of things - which are going to seem a bit rude. Sorry.

- PowerShell Remoting has been around since 2008. If you're seriously just asking yourself this now, then you're doing a poor job of managing your IT environment. Also released since 2008 were smart watches, Microsoft (formerly Windows) Azure, the Tesla Roadster, Disney's "Frozen," affordable LED light bulbs, and the iPhone 3G, 3GS, 4, 4S, 5, 5S, and 6 models. Just in case you missed those as well.
- Information technology is an industry of change. The perfectly reasonable decisions you made in 2003 are going to need to be periodically revisited, due to aforementioned change.

With that out of the way, let's briefly talk about...

What is PowerShell Remoting?

Remoting is simply a way for management tools on one computer to talk to services on another computer, so that you can remotely manage those services.

Remoting is based on HTTP, and uses a protocol called WS-Management (WS-MAN). It requires servers to have a single open port, and routes all incoming management traffic through that one port. WS-MAN traffic can be logged, can be proxied through security servers (provided by third parties), and can be completely encrypted by means of SSL.

Like all HTTP traffic, Remoting is essentially transmitting text back and forth. Remoting simply specifies a way for that text to be laid out (predominantly in an XML variant) so that tools and services can understand each other.

Remoting does not in any way affect the security of your network under default conditions. It does not, by default, transmit usernames or passwords - encrypted or otherwise. In a non-domain environment, you can create an environment where passwords would be transmitted, but it really wants that to be encrypted by SSL, so you'd be using HTTPS.

Remoting doesn't in any way give anyone special privileges. The technology literally can't let someone do something they don't have permission to do, unless you've gone through a rather complex setup for something called *delegated administration*. In that case, you can

enable specific users to perform specific tasks that they wouldn't normally be able to - but only through the specific channel and interface you've set up.

Comparing Remoting to what came before

Before Remoting, most Windows remote management was conducted over Remote Procedure Calls, or RPCs. These usually employed port-hopping, making them incredibly difficult to manage through a firewall. Contrast that with the one port you have to deal with in Remoting.

A lot of organizations today simply install management tools on servers, and then use Remote Desktop to "remotely manage." That's an incredibly poor idea, and is a major reason why Windows Server take so many patches, so many reboots, and other maladies. The code needed to support a full GUI, and therefore RDP, is massive - and that means patches are inevitable. Running management tools *on the server*, usually under administrator credentials, is opening the door to an attack.

In short, most organizations seem to be worred about the security "implications" of PowerShell Remoting. The implications are **significant** - Remoting is **much** more secure than what you've been doing. It's also more efficient, imposes less overhead on servers, and lets servers run with a leaner operating system that will require fewer patches, fewer reboots, and fewer service packs. Those servers will also boot faster, run fewer processes, run fewer services, and consume less storage space for the OS. Those servers will require less memory overhead for the OS, meaning you can pack more of 'em into a virtualization host. Sounds horrible, right?

But here's the best reason

Quite simply, the main reason to enable Remoting is that *you haven't got any choice.* Microsoft has moved firmly in this direction, and enables Remoting by default on Windows Server 2012 and later. The company *disables* Remote Desktop by default, which should tell you something.

Further, in Nano Server, *logging onto the console isn't even an option*, whether you use Remote Desktop or not. There *is* no local login. Remoting *is literally your only option* for managing the server. That's going to be the case going forward.

Running a Windows environment without enabling Remoting - at least on servers - is like driving a car without wanting to depress the accelerator. It isn't much fun, and you aren't going to get very far.